IE 498 Spring 2020 - Homework 5

Julius Olson

April 2020

1 Backpropagation

(a) (i)

$$\frac{\partial \rho}{\partial f} = \frac{\partial (y - f(x;\theta))^2}{\partial f} = -2(y - f(x;\theta)) = 2(f(x;\theta) - y) = \delta^4 \in \mathbb{R}$$

(ii)

$$\frac{\partial \rho}{\partial b^3} = \frac{\partial \rho}{\partial f}\frac{\partial f}{\partial b^3} = \delta^4 \cdot 1 = \delta^4 \qquad \in \mathbb{R}$$

$$\frac{\partial \rho}{\partial W^3} = \delta^4 \left(H^2\right)^T \qquad \in \mathbb{R}^{1 \times d_h}$$

(iii)

$$\frac{\partial \rho}{\partial Z^2} = \frac{\partial \rho}{\partial f}\frac{\partial f}{\partial H^2}\frac{\partial H^2}{\partial Z^2} = \left(W^3\right)^T \delta^4 \odot \sigma'(Z^2) = \delta^3 \in \mathbb{R}^{d_h \times 1}$$

(iv)

$$\frac{\partial \rho}{\partial b^2} = \frac{\partial \rho}{\partial Z^2}\frac{\partial Z^2}{\partial b^2} = \delta^3 \cdot 1 = \delta^3 \qquad \in \mathbb{R}^{d_h \times 1}$$

$$\frac{\partial \rho}{\partial W^2} = \frac{\partial \rho}{\partial Z^2}\frac{\partial Z^2}{\partial W^2} = \delta^3 \left(H^1\right)^T \qquad \in \mathbb{R}^{d_h \times d_h}$$

(v)

$$\frac{\partial \rho}{\partial Z^1} = \frac{\partial \rho}{\partial Z^2}\frac{\partial Z^2}{\partial H^1}\frac{\partial H^1}{\partial Z^1} = \left(W^2\right)^T \delta^3 \odot \sigma'(Z^1) = \delta^2 \in \mathbb{R}^{d_h \times 1}$$

(vi)

$$\frac{\partial \rho}{\partial b^1} = \frac{\partial \rho}{\partial Z^1}\frac{\partial Z^1}{\partial b^1} = \delta^2 \cdot 1 = \delta^2 \qquad \in \mathbb{R}^{d_h \times 1}$$
$$\frac{\partial \rho}{\partial W^1} = \frac{\partial \rho}{\partial Z^1}\frac{\partial Z^1}{\partial W^1} = \delta^2 (x)^T \qquad \in \mathbb{R}^{d_h \times d}$$

(b)
- For iteration $k = 0, 1, ...$
  Select a data sample (x, y) at random from $\left(x^i, y^i\right)_{i=1}^M$
- Calculate the gradients $G^k = \nabla_\theta \rho(f(x; \theta^k), y)$
- Update the parameters: $\theta^{k+1} = \theta^k - \alpha^k G^k$
- In this case this amounts to:
-

$$\begin{aligned}
\left(b^3\right)^{k+1} &= \left(b^3\right)^k - \alpha^k \delta^4 \\
\left(W^3\right)^{k+1} &= \left(W^3\right)^k - \alpha^k \delta^4 \left(H^2\right)^T \\
\left(b^2\right)^{k+1} &= \left(b^2\right)^k - \alpha^k \delta^3 \\
\left(W^2\right)^{k+1} &= \left(W^2\right)^k - \alpha^k \delta^3 \left(H^1\right)^T \\
\left(b^1\right)^{k+1} &= \left(b^1\right)^k - \alpha^k \delta^2 \\
\left(W^1\right)^{k+1} &= \left(W^1\right)^k - \alpha^k \delta^2 (x)^T
\end{aligned}$$

2 Stochastic Gradient Descent

(a) When using gradient descent to train a neural network we are only guaranteed to reach a local minimum. As there often are many local minima, there exists many different parameters θ that we could end up with at convergence. There is no guarantee that these parameters (θ^1 and θ^2) in this case will be similar.

(b) The theorem only states that the approximation error ϵ will be achieved when reaching a global minimum of the objective function. When using stochastic gradient descent we are however not guaranteed to reach a global minimum due to the non-convexity of the function. Thus, the SGD approach can result in reaching a local minimum at convergence. In the case that this happens an approximation within ϵ will not be achieved.

(c) The vanishing gradient problem refers to the fact that the gradients in the lower layers ($l << L$) of a neural network decrease in magnitude as the number of layers L grows. Therefore the problem is especially noticeable in deep neural models. As the gradients become smaller, the

convergence rate decreases drastically as each step updates the parameters of the lower layers at a slower rate. The problem can also occur due to saturation, i.e. the input of the hidden units is to large as $\lim_{||z||\to\infty} \sigma'(z) = 0$.
One solution is to use residual blocks in the network. In such block connections to the earlier layers in the network is provided and the output of a block is denoted as $F(x) + x$. Thus the derivative of the block will be larger as the residual connection does not pass through the activation function. Another solution is to use another type of activation function that doesn't squash the derivate to small values. An example of such a function would be the ReLU.

(d)

$$F_{softmax}(\mathbf{z}) = \frac{1}{\sum_{k=0}^{K-1}} \left(e^{z_0}, e^{z_1}, ..., e^{z_{K-1}}\right)$$

(e)

$$\sum_{k=0}^{\infty} \alpha_k = \infty \tag{1}$$

$$\sum_{k=0}^{\infty} (\alpha_k)^2 < \infty \tag{2}$$

The two requirements above have to be fulfilled by the learning rate in order to reach convergence. k refers to the iteration. An example of a learning rate function that meets these requirements is $\alpha_k = \frac{C_0}{C_1+k}$, where C_0, C_1 are constants.

(f) (i) **Dropout** (reduces model complexity):

$$R = \left\{R^1, ..., R^L\right\},\ R^l \in \mathbb{R}^N,\ R_i^l \sim Ber(p)$$
$$Z^l = W^l H^{l-1} + b^l$$
$$H^l = R^l \odot \sigma(Z^l)$$
$$f(X, R, \theta) = F_{softmax}\left(W^{L+1} H^L + b^{L+1}\right)$$

R^l is the mask for layer l and is applied by piecewise multiplication. As R_i^l are independent Bernoulli random variables, they either take the value 0 or 1. As such the mask removes some of the hidden units in the layer, and the complexity is thus reduced.

(ii) **L2 Regularization**: The cost function is altered by penalizing large values of weights. The result is reduced overfitting. The second term in the formula below is the L2 regularization term.

$$Cost = \rho(f(X;\theta), Y) + \lambda||\theta||^2$$

3 Training Algorithms

(a) RMSProp is an example of an adaptive learning rate algorithm, where the learning rate of each step is dependent on the learning rate of the previous step and the current gradient. The

algorithm is defined as:

$$r^l = \rho r^{l-1} + (1-\rho)(g^l)^2$$
$$\theta^l = \theta^{l-1} - \frac{\eta}{\sqrt{r^l + \epsilon}} \odot g^l$$
$$g^l = \nabla_\theta \mathcal{L}(\theta^l)$$

(b) Using standard SGD with a constant learning rate, two problems can occur. If the gradient is steep, each step taken will be quite far and thus the risk of overshooting the optimum is present. In regions where hte gradient is flat, the opposite problem occurs as the algorithm results in short step and thus a very slow convergence rate. RMSProp adapts the learning rate to the gradient in order to counteract these problems, taking smaller steps when the gradient is large and vice versa.

(c) Deep Q-Learning is an example of reinforcement learning. There exists a set of states X, a set of actions A applicable in each state by which the agent transitions from state to state. The purpose of the reinforcement learning is to assign the agent a numerical value called reward for executing a certain action a in a given state X_t. To select the optimal action at each state, a strategy $A_t = g(X_t)$ has to be learned. In deep q-learning a deep neural network is utilized to model/approximate the Q-value function to determine the optimal action. The problem becomes a regression problem with objective function defined as:

$$J(\theta_t) = (Y_t - Q(X_t, A_t; \theta_t))^2$$
$$Y_t = R(X_t, A_t) + \gamma Q\left(X_{t+1}, \arg\max_a Q(X_{t+1}, a; \theta_t); \theta_t\right)$$

(d) The ϵ-Greedy algorithm is commonly used.

$$A_t = \begin{cases} Uniform(a_1, ..., a_k) & \text{with probability } \epsilon \\ \arg\max_{a \in \mathcal{A}} Q(X_t, a'; \theta) & \text{with probability } 1-\epsilon \end{cases}$$

As the the number of epochs increases, ϵ is usually decreased in order to more often take the greedy action given by the model.

4 Initialization and Normalization

(a)

$$\mu_B = \frac{1}{M}\sum_{i=1}^{M} Z_i^l$$
$$\sigma_B^2 = \frac{1}{M}\sum_{i=1}^{M} \left(Z_i^l - \mu_{B,i}\right)$$
$$\hat{Z}_i^l = \frac{Z_i^l - \mu_{B,i}}{\sqrt{\sigma_{B,i}^2 + \epsilon}}$$
$$BN_\theta\left(Z^l\right) = \gamma \odot \hat{Z}^l + \beta$$

The algorithm above explains how each layer is normalized using statistics from the minibatch (of size M). The parameters γ and β in the last step are additional, and can be learned as well.

(b) While batch normalization obtains its normalization from a batch of inputs, layer normalization normalizes across the dimensions of a hidden layer for a single data sample. It is thus an exact normalization instead of an approximate, which is the result of batch normalization.

(c) Xavier Initialization is a way to initialize the weights of a layer in order to achieve a mean of 0 and variance of 1 for the outputs z_i in the layer. This is done in order to combat saturation of the hidden units. Each weight is initialized as a random variable according to (d_{in} refers to the dimension of the input):

$$w_i \sim \mathcal{N}\left(0, \frac{1}{d_{in}}\right)$$

(d)

$$x \in \mathbb{R}^d$$
$$Z = Wx + b^1$$
$$H = ReLU(Z)$$
$$f(x;\theta) = C^T H + b^2$$
$$\rho\left(f(x;\theta), y\right) = \left(y - f(x;\theta)\right)^2$$

Let C, W and b in the network above be initialized with only zeros. The backpropagation is

as follows (not all terms included for simplicity):

$$\frac{\partial \rho}{\partial f} = -2(y - f(x;\theta))$$
$$\frac{\partial \rho}{\partial C} = \frac{\partial \rho}{\partial f} H^T$$
$$\frac{\partial \rho}{\partial H} = \delta = C^T \frac{\partial \rho}{\partial f}$$
$$\frac{\partial \rho}{\partial b^1} = \delta \odot ReLU'(Z)$$
$$\frac{\partial \rho}{\partial W} = (\delta \odot ReLU'(Z))\, x^T$$

If all weight are initialized as zero, the output in the first layer (Z) will also be zero. This leads to the gradient w.r.t to C also being equal to zero. The same goes for δ and the gradient w.r.t b^1 and W. As such none of the parameter's value will change during backpropagation and thus the network will not train.

(e) Using ensemble models is a good way to reduce variance in the resulting model by averaging the output/prediction of several models. In this case the averaging is done over the parameters which are then used in a "new" model. Therefore the formula is not a mathematically correct ensemble model and should be altered as follows:

$$g(x) = \frac{1}{M} \sum_{m=1}^{M} f(x;\theta^m)$$

5 Convolutional Networks

(a) Consider a convolution between a kernel $K \in \mathbb{R}^{k_y \times k_x \times C_{out} \times C_{in}}$ and a padded input $\hat{X} \in \mathbb{R}^{d_y+2P \times d_x+2P \times C_{in}}$. C_{in} and C_{out} represent the number of input and output channels respectively, k_y and k_x are the filter dimensions and d_y and d_x are the dimensions of the input for one channel. P is the number of zeros added to each side of the input. The resulting output is denoted as: $Z_{:,:,p} = \hat{X}_{:,:} * K_{:,:,p}$. Given a stride length of s, the operation is formulated as:

$$Z_{i,j,p} = \sum_{p'=0}^{C_{in}-1} \sum_{m=0}^{k_y-1} \sum_{n=0}^{k_x-1} K_{m,n,p,p'} \hat{X}_{is+m,js+n,p'}$$

The output (Z) will be of the following dimensions:

$$\left(\lceil \frac{d_y - k_y + 2P}{s} + 1 \rceil \right) \times \left(\lceil \frac{d_x - k_x + 2P}{s} + 1 \rceil \right) \times C_{out}$$

(b) Using convolutions we are able to extract important features from e.g. images using much fewer parameters than fully connected, due to the parameter sharing that is a result of the

convolutional operation. The resulting convolutional models are thus much more efficient to train than their fully connected counterpart (both regarding computational and memory efficiency). Furthermore, the convolutional operation is invariant (equivariant) to translation. This means that the result of a convolution over an image that has been subjected to a translative function is the same as if one were to translate the output of the convolution over the original image with the same function. Convolutions can be thought of as a way of filtering the input in order to extract the useful information and filter out the unimportant aspects of the input.

(c)
- Stride (s): This hyperparameter equals the step size with which each filter moves over the input. A larger stride results in more downsampling as the overlap between receptive fields of the convolution decrease, and therefore the output will be of smaller dimensions.
- Padding (P): The amount of zeros that are added to each side of the input. Without padding, the dimensions of the output decrease for each convolutional layer that is applied in the network. By padding the inputs, we are able to preserve more of the input volume and thus are able to apply more layers and therefore get a more accurate analysis of the input.
- Channels (C_{in}, C_{out}): The number of the channels of the initial input of the network is dictated by which type of data is used. E.g. for a color image, there are one channel for each color. The output channels can be considered as a stack of feature maps in the resulting hidden layer. Increasing the amount of feature maps allow for the detection of more complex features in the input.

(d) The output dimensions are given by the following formula:

$$\left(\lceil\frac{d_y - k_y + 2P}{s} + 1\rceil\right) \times \left(\lceil\frac{d_x - k_x + 2P}{s} + 1\rceil\right) \times C_{out}$$

$$= \left(\frac{30-3}{1} + 1\right) \times \left(\frac{30-3}{1} + 1\right) \times C = 28 \times 28 \times C$$

(e)

$$I \in \mathbb{R}^{C_{in} \times d_x \times d_y \times d_z},\ K \in \mathbb{R}^{C_{out} \times C_{in} \times k_x \times k_y \times k_z}$$

$$Z_{p,i,j,q} = \sum_{p'=0}^{C_{in}-1} \sum_{m=0}^{k_y-1} \sum_{n=0}^{k_x-1} \sum_{r=0}^{k_z-1} K_{p,p',m,n,r} I_{p',i+m,j+n,q+r}$$

6 Pytorch and Implementation Questions

(a) Using float16 instead of float32 would enable larger models (and datasets) to be stored in memory on the CPU/GPU as each value uses less of the available memory. Furthermore, the training would be faster as the needed arithmetic bandwidth would be reduced.

(b) The computational cost for the backpropagation for mini-batch gradient descent is defined as: $M\left[(L-1)(3d_H^2+d_H)+d_H(2d_H+d+3K+1)+2K\right]$ and therefore grows linearly when increasing the size of the mini batch (M). The required memory for the backpropagation is defined as $(L-1)(d_H^2+d_H)+d_H(d+K)+K+2Md_H$ and hence also grows linearly when increasing the batch size (M).

(c) This can be done by loading the dataset into memory in batches. In pytorch this can be achieved using a dataloader.

(d) `opimizer.step()` is used to update the parameters of the model according to the calculated gradients (stored in `param.grad`). If it is omitted, the parameters simply will not be updated and the model will not train.

(e) `optimizer.zero_grad()` Sets the gradients of the parameters to zero. This done as the gradients are otherwise accumulated for each parameters, and therefore takes up much memory.

(f) Either `tensor.to(device)` (if `device == "cuda"`) or simply `tensor.cuda()` moves the tensor to the GPU.

(g) `dist.all_reduce(g, op=dist.reduce_op.SUM); g /= float(N).` The first command gathers all the gradients and sums them, and the second divides with the number of nodes and thus the combination of the two results in the average.

(h) When calculating the gradient at a step, the loss is obtained by the sum of the loss at each previous step. Even though the gradient is only calculated at every $k\tau,\ k=1,...\infty$ time steps, the gradient at that point is dependent on the loss of all the precious steps $t=1,...,k\tau-1$ as truncation is not used. Therefore the complexity grows towards infinity at a linear rate.

(i) When using truncated backpropagation through time, the loss is calculated as:

$$\mathcal{L}(\theta^{(k)}) = \sum_{t=\tau k+1}^{(k+1)\tau} \rho(Y_t, \hat{Y}_t)$$

And the computational cost is therefore $\mathcal{O}(\tau)$.